Programming Camp Summer 2023

Marcelo Sena, Summer 2023

I am grateful to Parth Sarin, Ciaran Rogers, Diego Jimenez and Brian Higgins for sharing material.


IF YOU COULD ACTUALLY MEET THE EXPECTATIONS YOU SET
THAT WOULD BE GREAT

What to expect from the programming camp:

- Familiarize *all* students with typical tools of Python
- Provide a clear basis so that students can further learn by themselves
- Level the playing field (so it may be slow for some)
- Make you aware of what is out there (for when you need more RAM or need to 100x faster)

What NOT to expect from it:

- Full proficiency (sorry!) - we are always learning =)

One important ground rule:

- Work together and help each other!

Tentative Outline

- Day 1:
  - Why is coding important?
  - Getting started with Python + basics
- Day 2:
  - Object oriented programming
  - Useful libraries
  - Debugging
  - General programming advice, other languages and further resources
- Day 3:
  - Servers, text editors and command line
- Day 4:
  - Parallel + GPU computing
  - AI and Coding

Data usage has increased in Economics (Angrist et al 2017, AER PP)

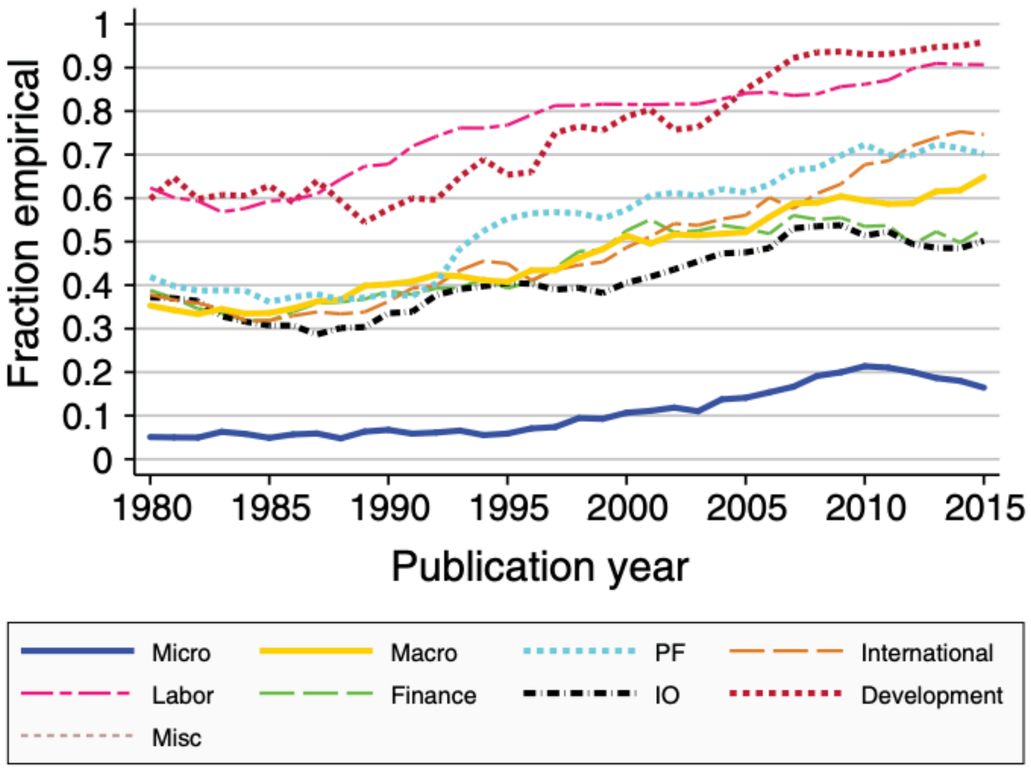


FIGURE 4. WEIGHTED FRACTION EMPIRICAL BY FIELD

Note: Five-year moving averages of the weighted fraction of publications in each field that are empirical.

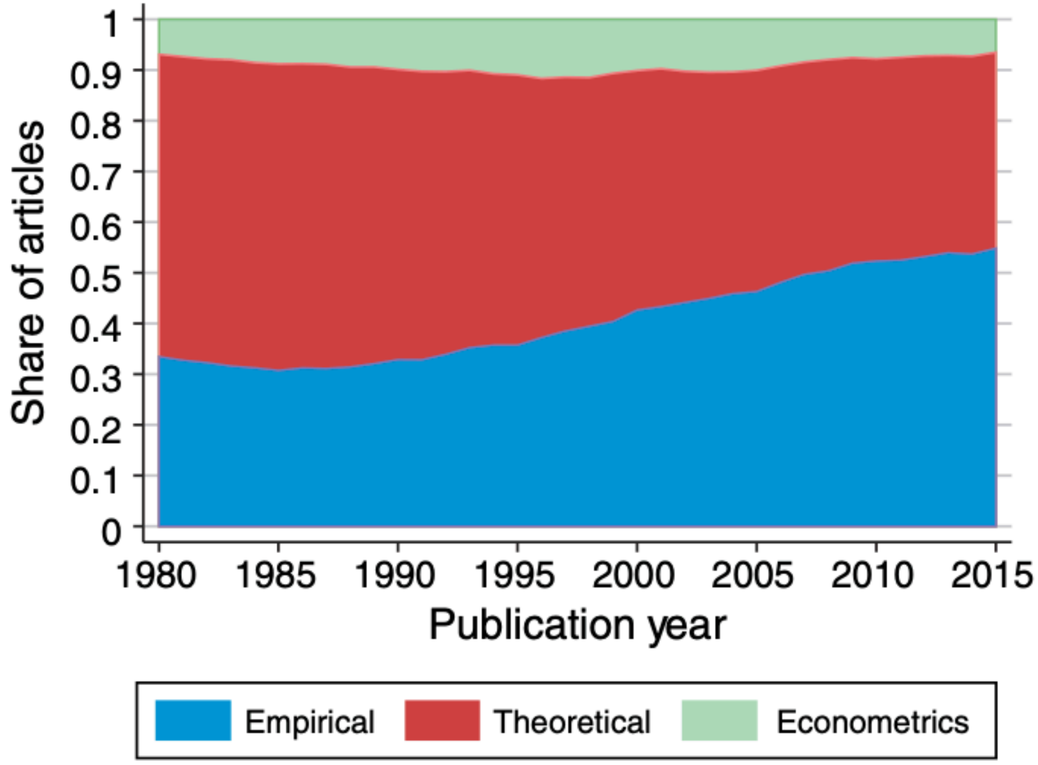


FIGURE 6. WEIGHTED PUBLICATIONS BY STYLE

Note: Five-year moving averages of weighted publication shares in each style.

Even among theorists!!! (Akbarpour, Malladi and Saberi (2020))

Just a Few Seeds More: Value of Network Information for Diffusion*

Identifying the optimal set of individuals to first receive information ('seeds') in a social network is a widely-studied question in many settings, such as diffusion of information, spread of microfinance programs, and adoption of new technologies. Numerous studies have proposed various network-centrality based heuristics to choose seeds in a way that is likely to boost diffusion. Here we show that, for the classic SIR model of diffusion and some of its generalizations, randomly seeding $s + x$ individuals can prompt a larger diffusion than optimally targeting the best s individuals, for a small x. We prove our results for large classes of random networks, and verify them in several small, real-world networks. Our results identify practically relevant settings under which collecting and analyzing network data to boost diffusion is not cost-effective.

Even among theorists!!!

A “Pencil-Sharpening” Algorithm for Two Player Stochastic Games with Perfect Monitoring

By Dilip Abreu, Benjamin Brooks, Yuliy Sannikov

April 28, 2016| Working Paper No. 3428

Economics

Download

We study the subgame perfect equilibria of two player stochastic games with perfect monitoring and geometric discounting. A novel algorithm is developed for calculating the discounted payoffs that can be attained in equilibrium. This algorithm generates a sequence of tuples of payoffs vectors, one payoff for each state, that move around the equilibrium payoff sets in a clockwise manner. The trajectory of these “pivot” payoffs asymptotically traces the boundary of the equilibrium payoff correspondence. We also provide an implementation of our algorithm, and preliminary simulations indicate that it is more efficient than existing methods. The theoretical results that underlie the algorithm also yield a bound on the number of extremal equilibrium payoffs.

Why Python?

- Popular! (most popular language according to many rankings)
  - Big community
- Free and open source
- General purpose: “second best language for everything”
- For economists: more uses outside macro problem sets!

Why Python?

ECONOMETRICA
JOURNAL OF THE ECONOMETRIC SOCIETY

Original Articles | Full Access

Using the Sequence-Space Jacobian to Solve and Estimate Heterogeneous-Agent Models

Adrien Auclert, Bence Bardóczy, Matthew Rognlie, Ludwig Straub

First published: 27 September 2021 | https://doi.org/10.3982/ECTA17434 | Citations: 1

Find it @ Stanford

For helpful comments, we thank four anonymous referees, as well as Riccardo Bianchi-Vimercati, Luigi Bocola, Michael Cai, Jesus Fernández-Villaverde, Joao Guerreiro, Kurt Mitman, Ben Moll, Laura Murphy, Martin Souchier, and Christian Wolf. Martin Souchier also provided outstanding research assistance. This research is supported by National Science Foundation Grant SES-1851717. This paper reflects our personal views and does not necessarily represent those of the Federal Reserve Board or the Federal Reserve System.

README.md

Sequence-Space Jacobian (SSJ)

SSJ is a toolkit for analyzing dynamic macroeconomic models with (or without) rich microeconomic heterogeneity.

The conceptual framework is based on our paper Adrien Auclert, Bence Bardóczy, Matthew Rognlie, Ludwig Straub (2021), Using the Sequence-Space Jacobian to Solve and Estimate Heterogeneous-Agent Models, Econometrica 89(5), pp. 2375–2408 [ungated copy].

Requirements and installation

SSJ runs on Python 3.7 or newer, and requires Python's core numerical libraries (NumPy, SciPy, Numba). We recommend that you first install the latest Anaconda distribution. This includes all of the packages and tools that you will need to run our code.

To install SSJ, open a terminal and type

```
pip install sequence-jacobian
```

Optional package: There is an optional interface for plotting the directed acyclic graph (DAG) representation of models, which requires Graphviz for Python. With Anaconda, you can install this by typing `conda install -c conda-forge python-graphviz`.

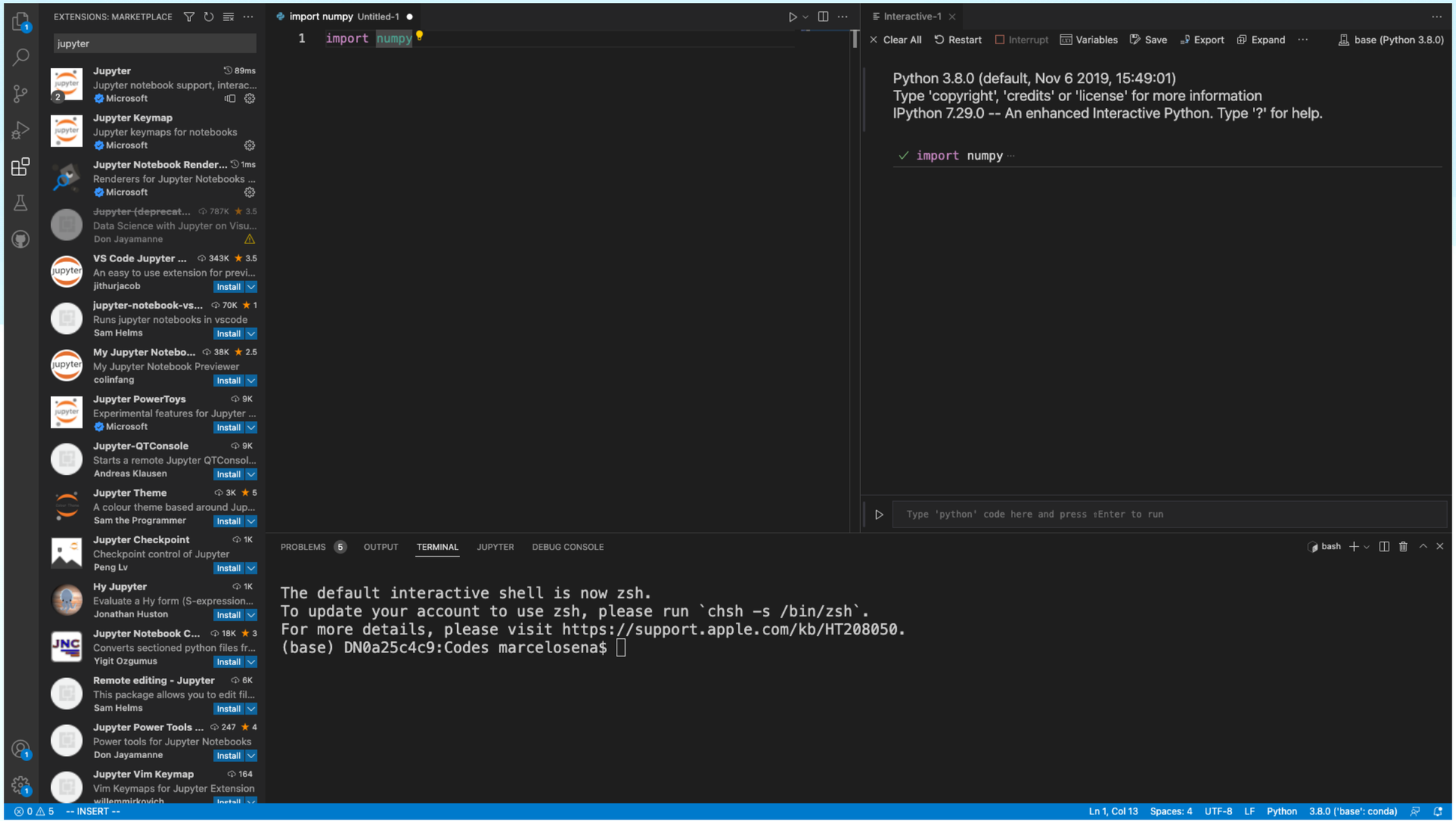
Getting started with Python

Setting up Python in your computer

- You don’t have to do this now!
- For programming camp, we will use Google Colab:
  - https://colab.research.google.com/
- To have Python in your computer, you can either:
  - Install the core Python package: https://www.python.org/downloads/
  - Install the Anaconda distribution: https://www.anaconda.com/products/enterprise
- Anaconda comes with Python + some extra Pythonic bells and whistles.
- Setting up overall workflow can be done in multiple ways and also a matter of personal taste.
- I recommend that you set up and experiment with this *before* Fall quarter starts.
  - Preferably during programming camp period so we can help you debug.

Setting up

What I use, for reference (VS Code + Jupyter Extension)



Setting up
Google Colab

- https://colab.research.google.com/
- Create a new notebook
- Make sure you can run the following

```
2+2
```

- You can run new code by either
  - Clicking the “play" button to the left of the cell
  - Pressing shift + enter (or return in a Mac)

Python Basics

Some basic syntax

Variables and Type

- Variables declaration in Python is done by

variable = value

- Python automatically infers the type of a variable from the type of the value.

- Variables can change type over the course of a program execution.

Some basic syntax

Variables and Type

- Know your types!
- E.g. documentation of the absolute function in Python.

abs(*x*)

Return the absolute value of a number. The argument may be an integer, a floating point number, or an object implementing `__abs__()`. If the argument is a complex number, its magnitude is returned.

- Typical number types: integers and floats.
- Typical text: strings

Some basic syntax

Variables and Types

- Different types occupy different memory sizes

Memory Size

Data Type	Memory Size
char	1 byte
int	4 bytes
float	4 bytes
double	8 bytes

Some basic syntax
Variables and Types

- Functions operate differently on different types

```
import numpy as np

array = np.array([1.0,2.0])
integer = 1
another_array =
np.array([1.0,2.0])
a_list = [1,2]
print(array + integer)
print(array + another_array)
print(array + a_list)
print(a_list + integer)
```

Some basic syntax

Math operations

- Standard mathematical syntax, except for exponentiation.

Numeric Operation	Operation Syntax	Assignment Syntax
Addition	`x + 5`	`x += 5`
Subtraction	`x - 5`	`x -= 5`
Multiplication	`x * 5`	`x *= 5`
Division	`x / 5`	`x /= 5`
Integer Division	`x // 5`	`x //= 5`
Modulo Operator	`x % 5`	`x %= 5`
Exponentiation	`x ** 5`	`x **= 5`

Boolean Operation	Operation Syntax
Not	`not a`
And	`a and b`
Or	`a or b`
Equals (Not Equals)	`a == b (a != b)`
Greater (Or Equal)	`a > b (a >= b)`
Less (Or Equal)	`a < b (a <= b)`
Chained Expressions	`a > b > c`

Some basic syntax

Math operations

- Bitwise operations: and (&), or (|)
  - Operates element by element on bit representation of numbers
  - e.g. 2 | 3 = 3
    - 2 in bit is 10
    - 3 in bit is 11
    - Return = [1 or 1][0 or 1] = 11 = 3 in bit representation
  - Conclusion: in practice, test multiple events, use and/or

Some basic syntax

Strings

```
game = "HHTTTHHHTHTT"
```

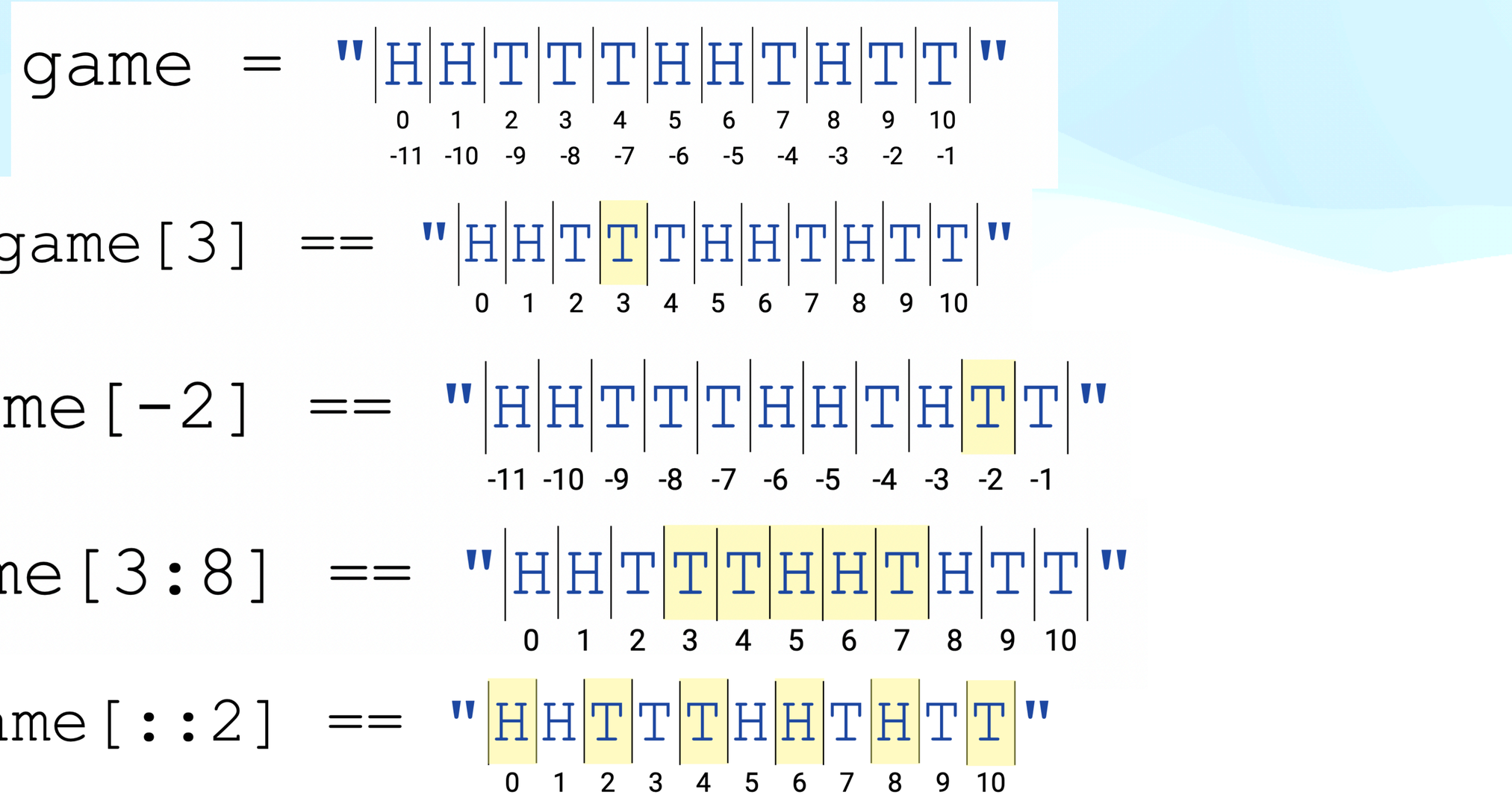
```
len(game)
```

```
"HTH" in game
```

Some basic syntax

Strings

IMPORTANT: THE FIRST ELEMENT IS INDEXED BY 0!



Some basic syntax

Tuples

- A tuple is a sequence type, meaning it stores an ordered collection of objects
- Immutable, meaning it cannot be changed after creation
- Stores heterogeneous data
- Packing…

```
congrats = ("Happy", 4, "you", "dude!")
```

Parentheses are conventional, but optional!

- Unpacking…

```
tup = (3, 2, 1)
three, two, one = tup
pow(*tup)
```

← The * indicates tuple unpacking

- Python understands the last expression as pow(3,2,1) = mod(3^2,1)

Some basic syntax

Lists

- A list is a mutable sequence type

```
psets_completed = ['macro',1,2]
```

- Special list methods include

`.count(elem)`	Counts the occurrences of `elem` in the list.
`.index(elem)`	Returns the index of the first occurrence of `elem` in the list.
`.append(elem)`	Appends the element `elem` to the end of the list.
`.extend(iterable)`	Extends the list by appending all elements of `iterable` to the end.
`.insert(idx, elem)`	Inserts the element `elem` at the index `idx` of the list.
`.sort(key=None, reverse=False)`	Sorts the list in-place.
`.reverse()`	Reverses the list in-place.
`.pop(i=-1)`	Returns and removes the ith element from the list.
`.remove(elem)`	Removes the first instance of `elem` from the list, or raises `ValueError`.

- This is object-oriented programming!

Some basic syntax

Lists

- Consider

```
tuple_lists = (['list1'],['list2'])
tuple_lists[0].append('another_item')
```

- Why were we able to change a tuple element?

Some basic syntax

Lists

- Consider

```
tuple_lists = (['list1'],['list2'])
tuple_lists[0].append('another_item')
```

- Why were we able to change a tuple element?
- Tuples stores references to underlying objects.
  - If the objects being referenced are mutable, they can still be changed.

Some basic syntax

A detour on assignment, copy and deep copies

- Assignments for sequence types in Python are done by reference.
- This means that assignments does not create copies, but only alias (a new name) for the same existing object.
- The following example illustrates this:

```
a = [1,2]
b = a
a[1] = 3
print(a)
print(b)
```

Some basic syntax

A detour on assignment, copy and deep copies

- For *soft* copies, we can use the copy function from the copy module.

```
import copy
a = [1,2]
b = copy.copy(a)
a[1] = 3
```

- If the sequence types contain other sequence type, then we need *deep* copies

```
a = [1,['list element 1','list element 2']]
b = copy.deepcopy(a)
a[1][0] = 'new value'
```

Some basic syntax

Dictionaries and Range

- Other sequence flavor types (dictionaries are actually Mapping types)
- Dictionaries provides key value pairs.

```
dic = {'key_1' : 'value_1', 'key_2' : 3, 'key_3' : ['a list']}
```

- Ranges represent a sequence of numbers typically used in for loops.

```
a_range_type = range(10)
```

Some basic syntax

Loops

- Python has both for and while loops.
- The beginning and end of loop blocks are defined with identation!
- Example

```
for i in range(10):
    print(i)
```

- The same is true for if statements

```
a_condition = True
if a_condition:
    print('a_condition is True')
else:
    print('a_condition is False')
```

Some basic syntax

Loops + Lists = List Comprehension

- List comprehension is a nice syntactic sugar to create objects iteratively in lists.
- Example:

```
even_numbers_list = [i for i in range(10) if i % 2 == 0]
```

- A syntax also to write loops more concisely
  - not necessarily preferable though

```
fibonacci_list = []
[fibonacci_list.append(0) if i == 0 else fibonacci_list.append(1) if i == 1 else
fibonacci_list.append(fibonacci_list[i-1] + fibonacci_list[i-2]) for i in range(10)]
print(fibonacci_list)
```

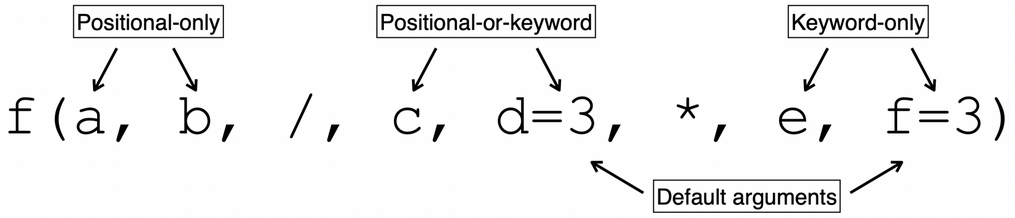
Some basic syntax

Functions

- As with loops, a function definition ends according to indentation.

```
def f(x1, x2):
    # Do stuff and save to x3
    return x3
```

- Function arguments can be positional or keyword and accepts default values.



Some basic syntax

Functions

- Functions can also take a varying number of arguments (variadic arguments)

```
def f_many_args(*objects):
    print(objects)

f_many_args(1,2,3)
f_many_args(*(1,2,3))
f_many_args((1,2,3))
```

Some basic syntax

Functions

- The first string inside a function body is taken to be its documentation string.

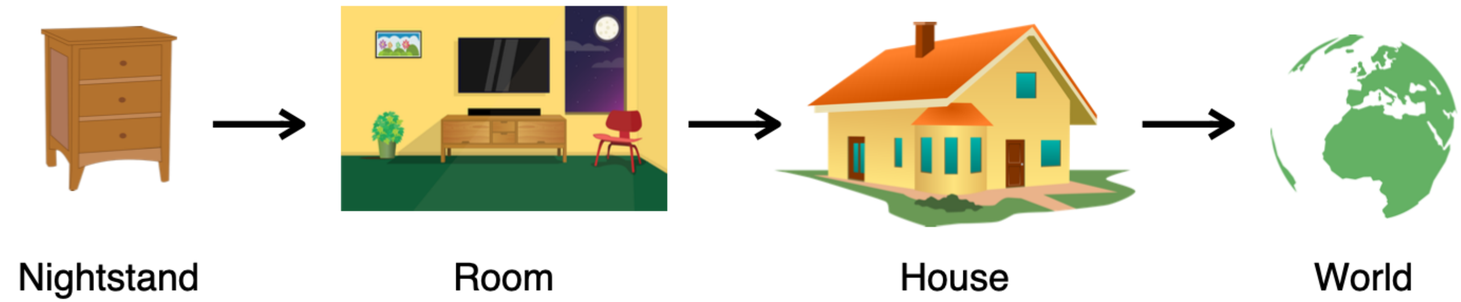
```
def f(x1, x2):
    """
    Description: Does some things.
    Arguments:
    - x1 (int): The first x.
    - x2 (int): The second x.
    Returns:
    - int: Integer representing the third x.
    """
    # Does some things
    return x3

print(f.__doc__)
```

Some basic syntax

Scoping

- Scoping refers to the rules that Python will follow to search for an object.
- Search goes from “inner to outwards”
  - think about how you would search for your lost keys



Local Scope → Enclosing Scope (if any) → Globals → Builtins

Some basic syntax

Scoping

- Example

```
def f():
    print(x)

def g(foo):
    x = 99
    foo()
    print(x)

x = 1
g(f)
```

- What is the output of g(f)?

Some basic syntax

Scoping

- Example

```
def f():
    print(x)

def g(foo):
    x = 99
    def a_function_defined_inside_a_function():
        print(x)
    foo()
    print("below will be the function defined inside g")
    a_function_defined_inside_a_function()
    print(x)

x = 1
g(f)
```

- What is the output of g(f)?

Some basic syntax

Scoping

- Example

```
def f():
    print(x)

def g(foo):
    x = 99
    def a_function_defined_inside_a_function():
        print(x)
    foo()
    print("below will be the function defined inside g")
    a_function_defined_inside_a_function()
    print(x)

x = 1
g(f)
```

- What is the output of g(f)?
- Conclusion: a scope search begins in the environment the function is defined!

Some basic syntax

Scoping

- What happens in the following example?

```
x = 1
def myfun():
    return x
x = 10
myfun()
```

Some basic syntax
Scoping

- What happens in the following example?

```
x = 1
def myfun():
    return x
x = 10
myfun()
```

- Conclusion: a function searches for a variable when it is called, not when it is defined

Object-Oriented Programming

Programming Paradigms

- A way to classify programming languages based on their features.
- A language can be classified into multiple of these.
- The common paradigm terminology you may encounter are two:
  - Functional programming -> R! (Remember maps)
  - Objected-oriented programming (OOP) -> Python!
- Most languages nowadays blends all of these paradigms.
  - Python is actually multi-paradigm.

Object-oriented Programming

- With OOP, data and functions are “bundled together” into objects.
- In Python, everything is an object, which consists of:
  - A type
  - An identity
  - Data
  - Methods

Object-oriented Programming

- High-level idea: create blueprint
- E.g. create a blueprint for creating a house.
  - houses are different, but they share many similarities
  - this makes it easy to build many houses.
- In economics: blueprint for heterogeneous agents

Object-oriented Programming

- We have already seen an instance of OOP in a previous example

```
print(f.__doc__)
```

- __doc__ is a method of a function f (which is itself an object).
- underscores have no syntactical meaning
  - just there to avoid method names clashes

Object-oriented Programming

- With OOP we can create our own objects and write methods for it!
- Let's write a class that is a class that defines a consumer.

```
class Consumer:

    def __init__(self, g, wealth):
        self.risk_aversion = g
        self.util = 0.0
        self.wealth = wealth

    def eat(self, c):
        # consumer's utility from consuming c
        self.util += c**(1-self.risk_aversion)/(1-self.risk_aversion)
        self.wealth -= c
```

Object-oriented Programming

Subclasses

- We can also define a subclass.
- Idea: create specialization of objects with specific methods
  - But still adheres still accepts the parents' methods
  - In computer-science lingo, this is called inheritance
- Think about a class for a motorized vehicle
  - we could then build a subclass for cars
- For our consumer class case, some consumers may have a borrowing constraint

Object-oriented Programming

Subclasses

```
# Parent Class
class Consumer:

    def __init__(self, g, wealth):
        self.risk_aversion = g
        self.util = 0.0
        self.wealth = wealth

    def eat(self, c):
        # consumer's utility from consuming c
        self.util += c**(1-self.risk_aversion)/(1-self.risk_aversion)
        self.wealth -= c
```

```
# Consumer subclass
class Consumer_Constrained(Consumer):

    def __init__(self, g, wealth):
        super(Consumer_Constrained, self).__init__(g,
                wealth)
        # super().__init__(g, wealth) # alternative syntax
        self.loan = 0

    def borrow(self, l):
        # consumer takes a loan l
        self.loan += l
        self.wealth += l

consumer1 = Consumer_Constrained(2, 10.0)
consumer1.eat(2)
print(consumer1.util)
print(consumer1.wealth)
consumer1.borrow(5)
print(consumer1.wealth)
```

Object-oriented Programming

- Some packages require you to write classes to leverage on its own methods.
- More involved example from PyTorch (deep learning framework).
- This is PyTorch first example in its tutorial.

```
import torch.nn as nn

class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )
```

This is a subclass example: super() tells Python for the class NeuralNetwork to inherit its methods from the base class (nn.Module)

Useful libraries

Useful libraries

- NumPy
- Matplotlib
- SciPy
- Pandas
- Sympy
- Numba

Useful libraries

Installing new packages

- If you need to install a new package, you should run either (depends on how you installed Python)

```
pip install numpy
```

```
conda install numpy
```

- In Python, you can also create virtual environments, which are essentially “local workspaces” which already contains the packages required (and its versions) to run the programs you desire.
  - These are essentially used to ensure consistency of versions of packages consistent when, for example, running programs across different different computers.
  - To create an environment

```
conda create —name <environment-name>
```

or

```
python -m venv <environment-directory>
```

- We won’t do any of this, but its useful to know it exists.

Useful libraries

Numpy

- A library for computations with array (think “Matlab in Python”)
- Native functions in NumPy are very efficient (written in C).
- You will probably need it for virtually any use of Python for your problem sets/ projects.

```
# Running a regression in numpy

import numpy as any_alias_for_numpy

x = any_alias_for_numpy.random.randn(100,1)
e = any_alias_for_numpy.random.randn(100,1)

y = 0.7*x + e

beta_hat = any_alias_for_numpy.linalg.inv(x.T @ x ) @ x.T @ y
```

Useful libraries

Numpy

- For the alias, people typically use

```
import numpy as np
```

- Numpy allows you to the array operations you would expect.
- If you don’t find a method/function that does what you want then you are probably trying to do it the wrong way.
- A few examples:

```
import numpy as np
data = np.array([[1, 2], [3, 4], [5, 6]]) # creates a 3x2 matrix
print(data[0,1]) # accesses (1,2) element of array
print(data[0,0:2]) # slicing arrays
print(data.max()) # maximum element of array
print(data.max(axis = 0)) # maximum element of array over rows
print(data.reshape(2,3)) # reshaping
```

Useful libraries

Plotting

- The main plotting library is Matplotlib.
- For basic usage, it is typically loaded as

```
import matplotlib.pyplot as plt
```

- Plotting a function on a grid

```
import matplotlib.pyplot as plt
import numpy as np

# make data
x = np.linspace(0, 10, 100)
y = 4 + 2 * np.sin(2 * x)
```

Useful libraries

Plotting

- For different plot types, use a different plotting method.
- Plotting our regression data as a scatter

```
import matplotlib.pyplot as plt
plt.scatter(x,y)
plt.show()
```

Useful libraries

Plotting

- The plot function has many arguments for customatization.

```
import matplotlib.pyplot as plt
import numpy as np

# make data
x = np.linspace(0, 10, 100)
y = 4 + 2 * np.sin(2 * x)

plt.plot(x,y,color = 'red')
plt.show()
```

Property	Description
agg_filter	a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array and two offsets from the bottom left corner of the image
alpha	scalar or None
animated	bool
antialiased or aa	bool
clip_box	Bbox
clip_on	bool
clip_path	Patch or (Path, Transform) or None
color or c	color

Useful libraries

Plotting

- You can customize axis, legends, titles, etc… with methods from the PyPlot object

```
import matplotlib.pyplot as plt
import numpy as np

# make data
x = np.linspace(0, 10, 100)
y = 4 + 2 * np.sin(2 * x)

plt.axes(xlabel = 'the x grid', ylabel = 'the value of the function')
plt.plot(x,y,color = 'red', label = 'a transformation of the sign function')
plt.legend()
plt.show()
```

```
matplotlib.pyplot
    matplotlib.pyplot
        matplotlib.pyplot.acorr
        matplotlib.pyplot.angle_spectrum
        matplotlib.pyplot.annotate
        matplotlib.pyplot.arrow
        matplotlib.pyplot.autoscale
        matplotlib.pyplot.autumn
        matplotlib.pyplot.axes
        matplotlib.pyplot.axhline
        matplotlib.pyplot.axhspan
        matplotlib.pyplot.axis
        matplotlib.pyplot.axline
        matplotlib.pyplot.axvline
        matplotlib.pyplot.axvspan
        matplotlib.pyplot.bar
        matplotlib.pyplot.bar_label
        matplotlib.pyplot.barbs
        matplotlib.pyplot.barh
        matplotlib.pyplot.bone
        matplotlib.pyplot.box
        matplotlib.pyplot.boxplot
```

Useful libraries

Plotting

- You can create subplots with the subplots method.

```
import matplotlib.pyplot as plt
import numpy as np

# Some example data to display
x = np.linspace(0, 2 * np.pi, 400)
y = np.sin(x ** 2)

fig, (ax1, ax2) = plt.subplots(2)
fig.suptitle('Vertically stacked subplots')
ax1.plot(x, y)
ax2.plot(x, -y)
```

Useful libraries

Plotting

- You can also use a style file

```
print(plt.style.available)
plt.style.use('fivethirtyeight')
```

Useful libraries
Plotting

- You can even create your own style file.
- Place it in ~/.matplotlib/stylelib/

```
# Figure Properties

figure.figsize: 7,7
font.family: Helvetica
font.size: 20
axes.linewidth: 2
axes.labelpad: 10
axes.labelsize: 24
text.usetex: True

# Shades of grey/black
axes.prop_cycle: cycler('color',['C5C9C7','929591','808080','000000'])
```

Useful libraries

SciPy

- Provides algorithms for optimization, integration, interpolation, differential equations, etc…
- An optimization example:

$$\max_{c_0,c_1} log(c_0) + 2log(c_1) \text{ s.t. } p_0c_0 + p_1c_1 = w$$

Useful libraries

SciPy

- An optimization example (note syntax will depend on the method you pick!):

```
from scipy.optimize import LinearConstraint, Bounds, minimize
import numpy as np

wealth = 10.0
p_c0 = 1.0
p_c1 = 1.5
bounds = Bounds([0.001,0.001],[np.inf, np.inf])
linear_constraint = LinearConstraint([[p_c0, p_c1]], [0.001], [wealth])

def utility(c):
    return -1*(np.log(c[0]) + 2*np.log(c[1]))

x0 = np.array([0.5, 0.5])
res = minimize(utility, x0, method='trust-constr', constraints=[linear_constraint],
               options={'verbose': 1}, bounds=bounds)
```

Useful libraries
Pandas (Dataframes)

- The main package to manipulate data in Python is pandas.
- The main object there is also a dataframe, which is the typical datatable with rows being observations and columns being different variables.
- In Python we typically do not use piping* syntax (as e.g. in R/tidyverse), but we can leverage dataframe methods.
- Download raw data for Apple, Microsoft, IBM and Uber balance sheet data.

https://github.com/marcelosena/programming_camp/blob/main/wrds_data.txt

*although possible, see pipe package.

Useful libraries
Pandas (Dataframes)

- Manipulating raw data:

```
# raw data manipulation
import pandas as pd
import matplotlib.pyplot as plt

data = pd.read_csv('../Data/wrds_data.txt', sep = "\t")

# quick look at the data
print(data.head())
# transform datadate variable into proper date
data['datadate'] = pd.to_datetime(data['datadate'], format = '%Y%m%d')
# make datadate the index
data = data.set_index('datadate')
## view available columns
print(data.columns)
# dropping add1 variable
data = data.drop('add1', axis = 1)
# dropping last observation
data = data.drop(index=pd.to_datetime('2020-06-30'))
# pandas dataframes have a plot method too
# grouping by companies and plotting total assets
data.groupby('conml')['actq'].plot(legend = True)
plt.show()
```

Useful libraries
Pandas (Dataframes)

- Other useful panda methods

```
data.merge(...) # merges data
data.pivot(...) # reshapes data
data.iloc[0,:] # selects the first row
data.iloc[:,0] # selects the first column
data.loc[:,'col1'] # selects the column with name 'col1'
data.shift(1) # shifts the data by one row (lags)
data.dropna() # drops all rows with missing values
```

Useful libraries
SymPy

- SymPy is a computer algebra package for Python.
- Useful for tedious algebra or double checking own derivations.

```
from sympy import *

# declaring symbolic variables
c = symbols('c', real = True)
v = Function('v')(c)
param = symbols('\gamma', real = True, positive = True)

obj = log(c) + param*v
c_foc = diff(obj, c)
c_sol = solve(c_foc, c)[0]
print(c_sol)
```

Useful libraries
Numba

- Numba is a package that allows “free” code speedup
- We will do this more in-depth in Econ 210
- Example:

```
import numba
import random
import time

start = time.time()
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples

print(monte_carlo_pi(10_000_000))
end = time.time()
print(end - start)
```

Useful libraries
Numba

- Speeding up: @numba.jit() is called a function decorator

```
start = time.time()
@numba.jit()
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples

print(monte_carlo_pi(10_000_000))
end = time.time()
print(end - start)
```

- Recommend setting @numba.jit(nopython = True) or the shorthand @numba.njit()
  - Guarantees weakly faster code (modulo debugging)

Useful libraries
Numba

- Once a function is compiled, global variables are “hard-coded” into function
- This will make functions seem to behave *differently* from Python’s standard scoping rules

```
import numba
import numpy as np

x = 1
def f():
    print(x)

@numba.njit()
def numba_f():
    print(x)
numba_f()
f()

x = 10
f()
numba_f()
```

Detour: function decorators

- A function decorator would be the coding equivalent of a functional/operator (a machine that takes a function as an argument)

```
import time
import math

def function_timer(func):
    #the inner1 function takes arguments through *args and **kwargs

    def inner1(*args, **kwargs):
        # storing time before function execution
        begin = time.time()

        func(*args, **kwargs)

        # storing time after function execution
        end = time.time()
        print("Total time to run " + func.__name__ + " function: " + str(end - begin))

    return inner1

@function_timer
def compute_model(num):
    print("Model computed succesfully!")

# calling the function.
compute_model(10)
```

- The decorator above will automatically make a function time itself

Useful libraries
Other libraries/Python related languages

- scikit-learn: machine learning + other statistical techniques
- PyTorch: deep learning
- Keras: deep learning
- Tensorflow: deep learning
- Mojo
- Jax

Debugging

Debugging

- Debugging may also involve some personal taste.
- Printing objects and running a code may work, but it is not the efficient way.
- Python has a native debugger which allows you to pause code at desired points of execution.
  - You can then inspect your variables at that point of the execution or run the code line by line.
- Interactive Development Environments will also typically have practical debugging options.

Debugging
Native debugger

- In notebooks, we can debug using the %debug command.
- To set a breakpoint, use the pdb package

```
import numpy as np
import pdb
def return_dimension(array):
    pdb.set_trace()
    # this will give an error because shape is a property of the array, not a method
    return array.shape()
print(return_dimension(np.array([[1, 2], [3, 4], [5, 6]])))
```

- To step the code line by line, enter n.
- For more commands use h.
- VS Code has a visual implementation of this

Getting help

- Obviously: google it
- Stack overflow: make sure its a good question!
  - Post a minimum working example
  - guides you to the core of the error and serves for others to replicate your error
  - more time-consuming, but during this process you might actually find the bug yourself!
- Look at the documentation
  - Typically not economist friendly, but computer science friendly
  - Take a deep breath and go over it slowly
- LLMs and Github Co-Pilot

General Programming Advice

Some coding advice

- Stanford is one of the best places in the world to learn Computer Science tools.
  - If interested, leverage that!
- Use problem sets to try out new things and learn new things.
  - If you already know how package X works, try out package Y.
  - If you are an expert in R, try doing a problem set in Python (or Julia!).
  - Preferably, pick those problem sets which you think will be easier for you.
- There are many ways you can code the same output, so I always start with whatever is easier
  - If I need to optimize, I will do it later (knowing what the correct output should be)
- Learn to read error messages

Some coding advice

Coding classes in Stanford

- CS106A - Python and general programming principles
- CS106B - C++ and general programming principles
- CME 193 - Introduction to Scientific Python
- CS229 - Machine Learning (all done in Python)
- CS230 - Deep Learning (all done in Python)
- CS224N - Natural Language Processing (all done in Python)